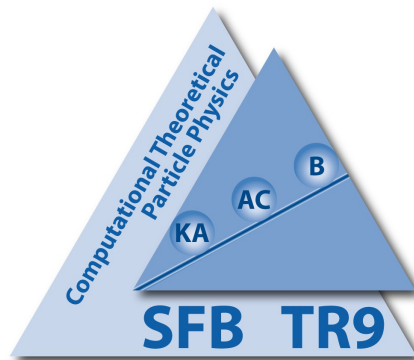


ParFORM and FORM4

Takahiro Ueda
TTP KIT Karlsruhe, Germany



International Workshop on
Frontiers in Perturbative Quantum Field Theory
11 September 2012, Bielefeld U.

Outline

- Introduction
- FORM and its parallel versions
- New features in FORM 4 (and also in parallel versions)
- Summary

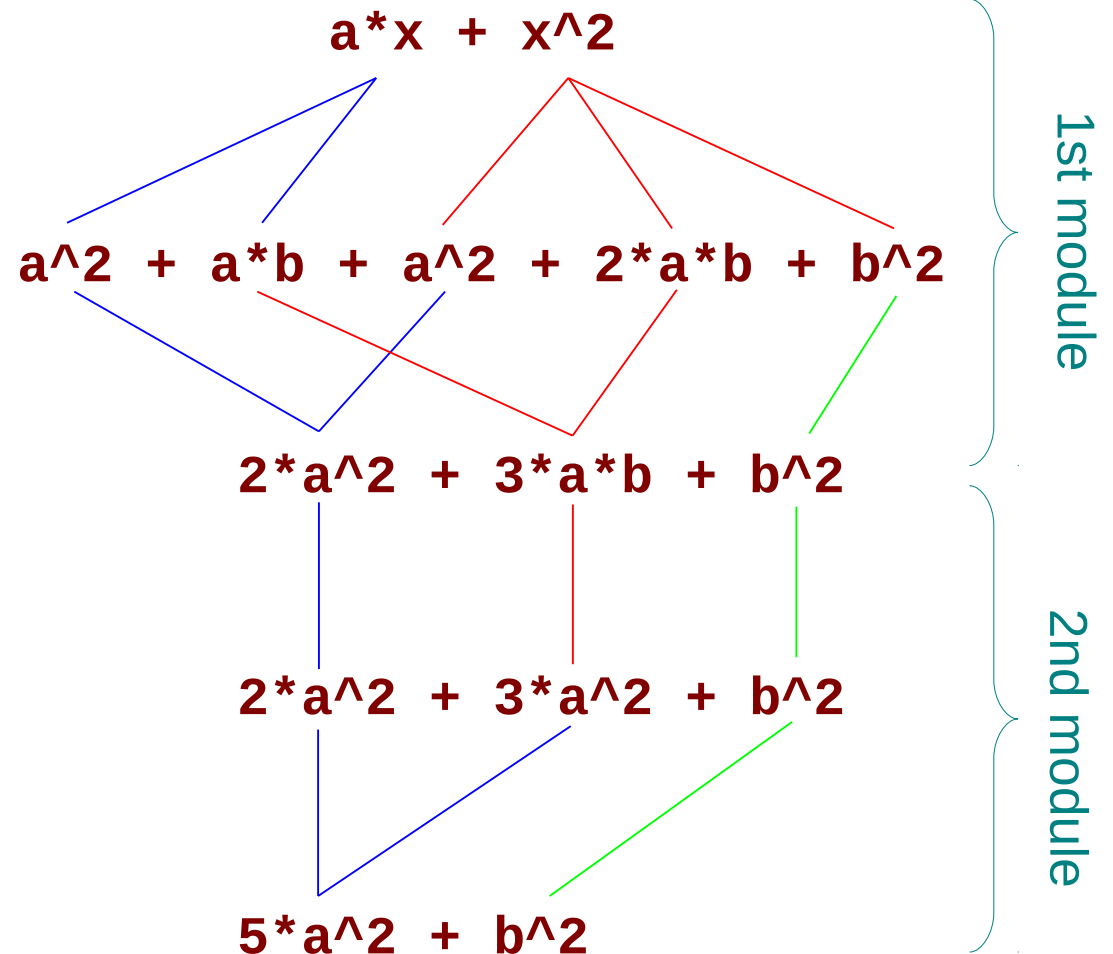
Introduction

- **FORM** (by J. Vermaseren et al.) is a program for symbolic manipulations, which can handle expressions consisting of a huge number of terms (\geq TB).
- The current version is FORM 4.0.
FORM version 4.0, J. Kuipers, TU, J.A.M. Vermaseren, and J. Vollinga, arXiv:1203.6543 [cs.SC].
- The FORM web site: <http://www.nikhef.nl/~form>
- Parallel versions, which make use of multiple CPUs simultaneously:
 - **ParFORM**: the Message Passing Interface (MPI).
 - **TFORM**: the POSIX threads (Pthreads).

The First Example

- User program:

```
Symbol a,b,x;  
Local expr = a*x + x^2;  
  
identify x = a + b;  
  
.sort  
  
if (count(b,1) == 1)  
    multiply a/b;  
  
Print;  
.end
```



FORM 4.0 (Sep 4 2012) 64-bits

Symbol a,b,x;

Local expr = a*x + x^2;

identify x = a + b;

.sort

Time =	0.00 sec	Generated terms =	5
	expr	Terms in output =	3
		Bytes used =	108

if (count(b,1) == 1)
multiply a/b;

Print;
.end

Time =	0.00 sec	Generated terms =	3
	expr	Terms in output =	2
		Bytes used =	64

expr =
b^2 + 5*a^2;

0.00 sec out of 0.00 sec

\$ form example.frm

More Examples: Pattern Matching

- For more complicated manipulations (differentiation, integration, series expansion, etc.), one need to use powerful and flexible features of FORM, e.g., **pattern matching**.

For example, differentiation of a polynomial with respect to x can be written as follows:

```
Symbol x,n;  
Local expr = 1 + x + x^2 + x^3;  
  
identify x^n? = n * x^(n-1);  
  
Print;  
.end
```

← Matches with x to the power of any integer n (including zero).

More Examples: Other Types of Objects

- FORM has many types of objects: symbols, functions, vectors, indices etc.

```
Symbol x, a;  
CFunction den;  
Local expr = den(1-2*x);
```

Series expansion of $\text{den}(a+b*x) = \frac{1}{a+bx}$
up to $\mathcal{O}(x^5)$ (with nonzero a and b).

```
#define N "5"
```

Split arguments as $\text{den}(a+b*x) \Rightarrow \text{den}(a,b*x)$

```
splitarg (x) den;  
repeat;  
  identify den(a?,x?) = 1/a - x/a * den(a,x);  
  if (count(x,1) > `N') discard;  
endrepeat;
```

```
Print;  
end
```

Repeat $\frac{1}{a+bx} = \frac{1}{a} - \frac{bx}{a} \frac{1}{a+bx}$

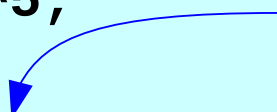
until enough higher terms are obtained.

More Examples: \$-variables

- \$-variables are basically small expressions. They can store various types of information and can be accessed in both compile-time (preprocessor) and run-time (processor).
- The following code is to determine the maximum power of x in expressions:

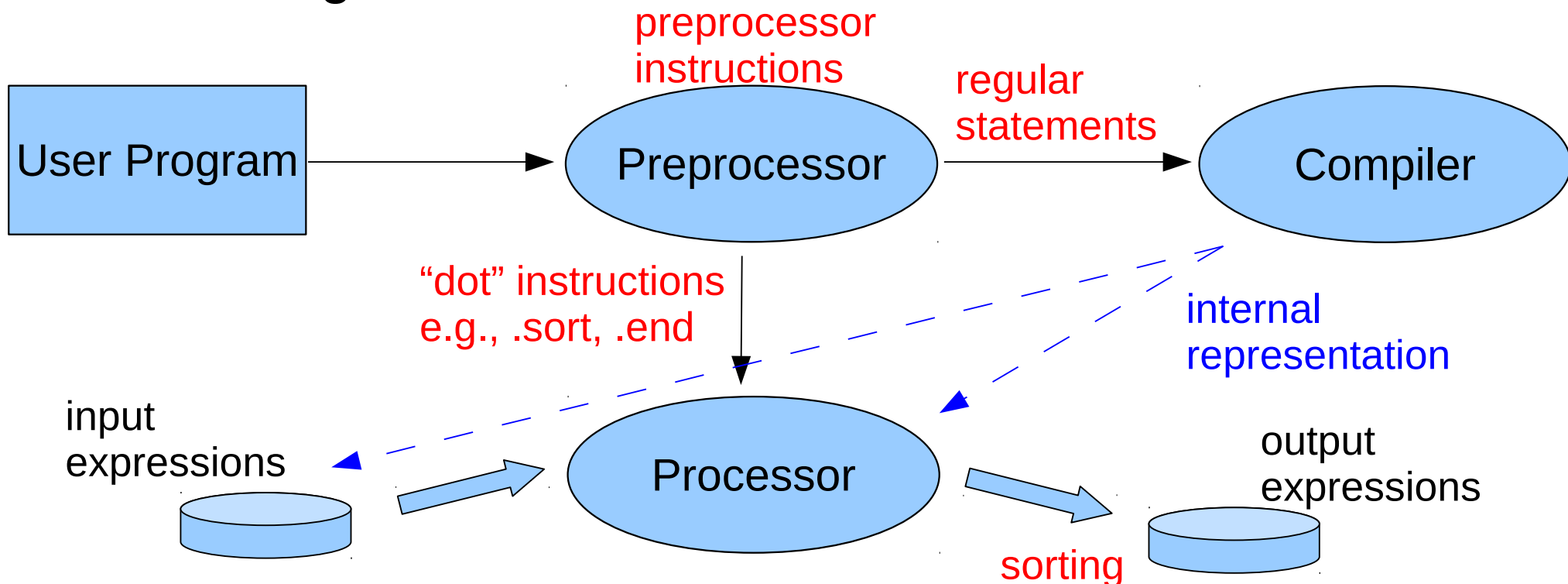
```
Symbol x,n;  
Local expr = (1+x)^5;  
#$n = 0;  
if ( count(x,1) > $n ) $n = count_(x,1);  
.sort  
  
#message the maximum power of x is ` $n '  
.end
```

If the higher power of x is found,
store it into \$n.



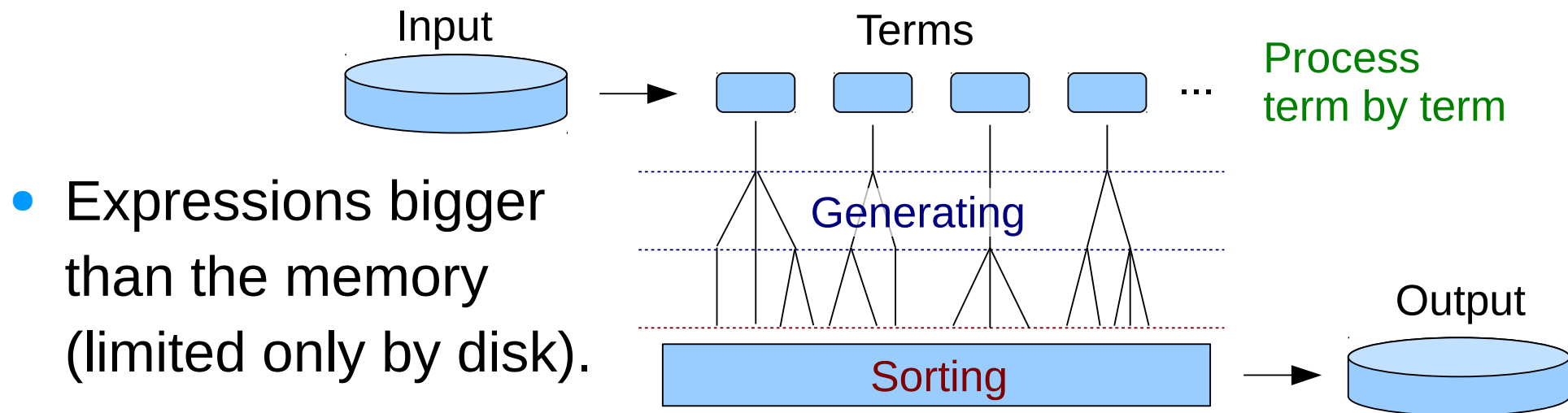
Structure of FORM

- Preprocessor: preparation and filter of the user input.
- Compiler: compiles statements to internal representation.
- Processor: execution of statements, generation of terms and sorting them.



Sequential FORM

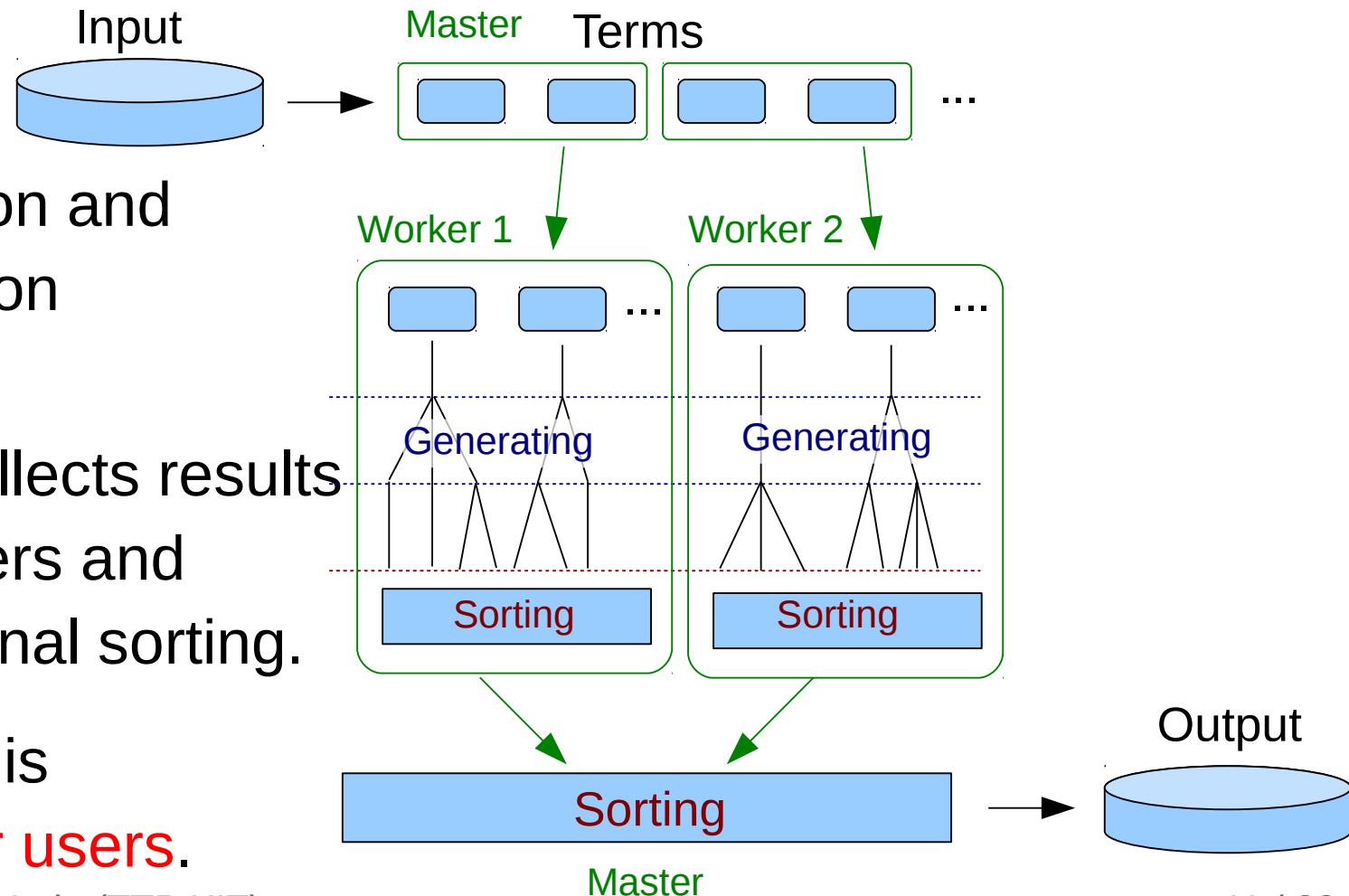
- Locality principle:
 - Operations are local for each term.
 - Complete data are stored locally for each term.
 - Can process **each term independently**.
- Expressions as “streams” of terms.
 - **Sequential access** to the disk storage. Merge sort on disk.



- Expressions bigger than the memory (limited only by disk).

Concept of Parallelisation of FORM

- Based on **master-worker model**.
- The master distributes terms to workers.



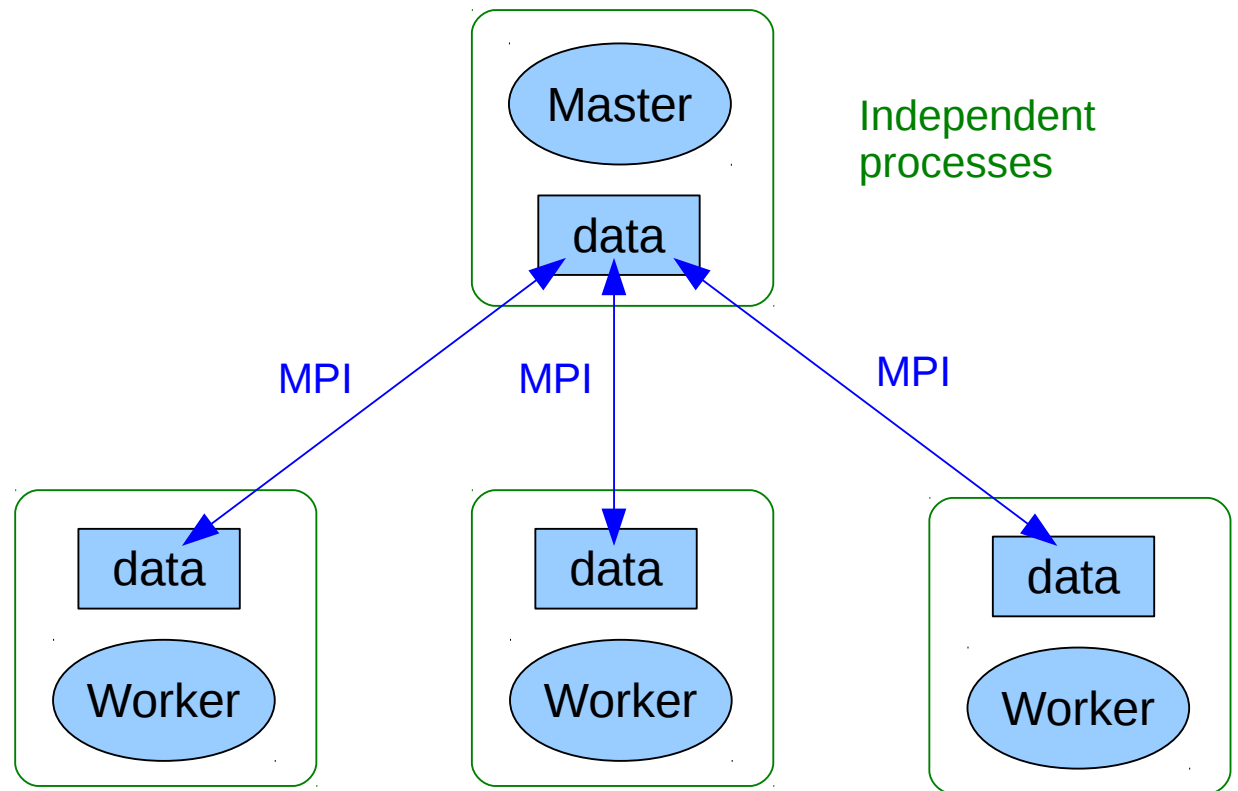
- Term generation and partial sorting on each worker.
- The master collects results from the workers and performs the final sorting.
- Parallelisation is **transparent for users**.

ParFORM

- Multiprocessor version of FORM.
- Communication via the Message Passing Interface (MPI).
- Can run on computer clusters.

Karlsruhe, 1998-

Fliegner, Retey, Vermaseren '00
Tentyukov, Fliegner, Frank,
Onischenko, Retey, Staudenmaier '04
Tentyukov, Staudenmaier,
Vermaseren '06

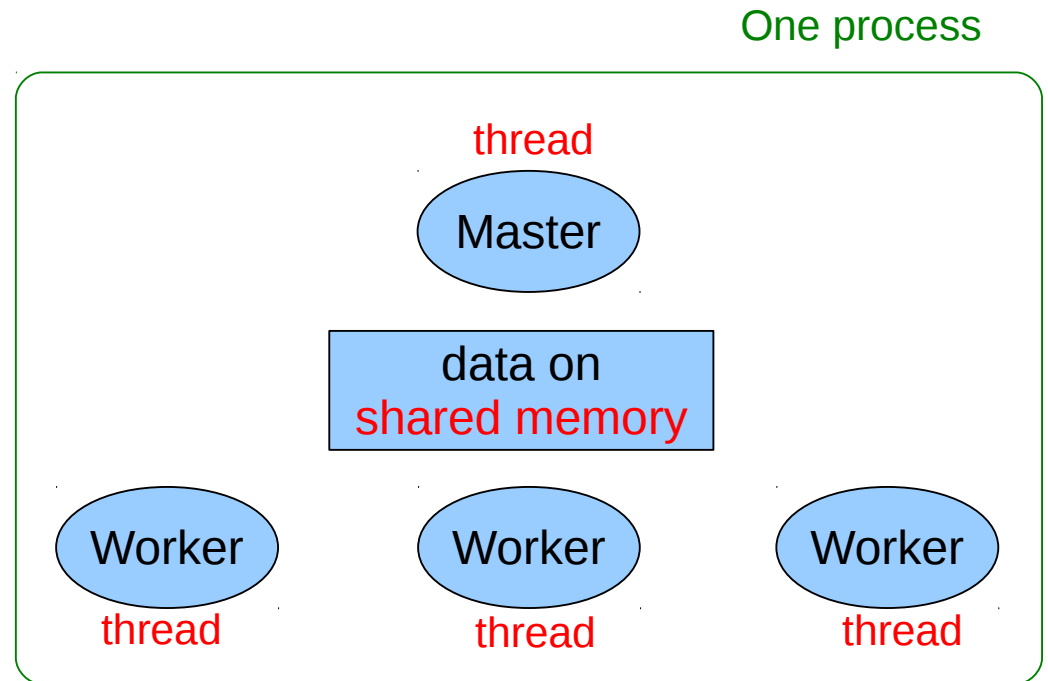


TFORM

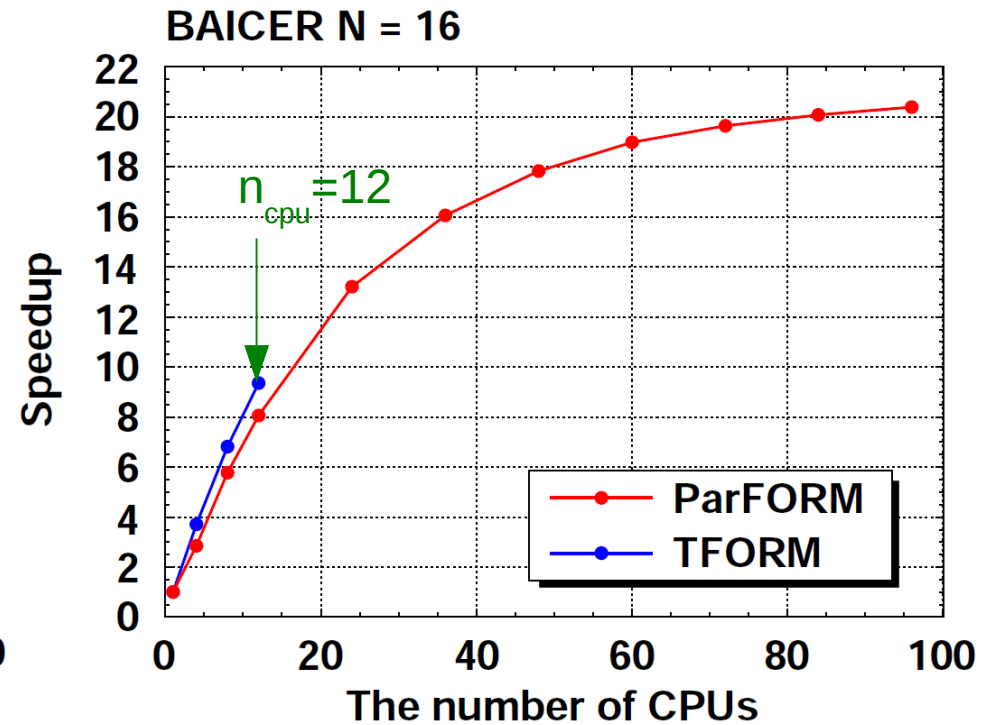
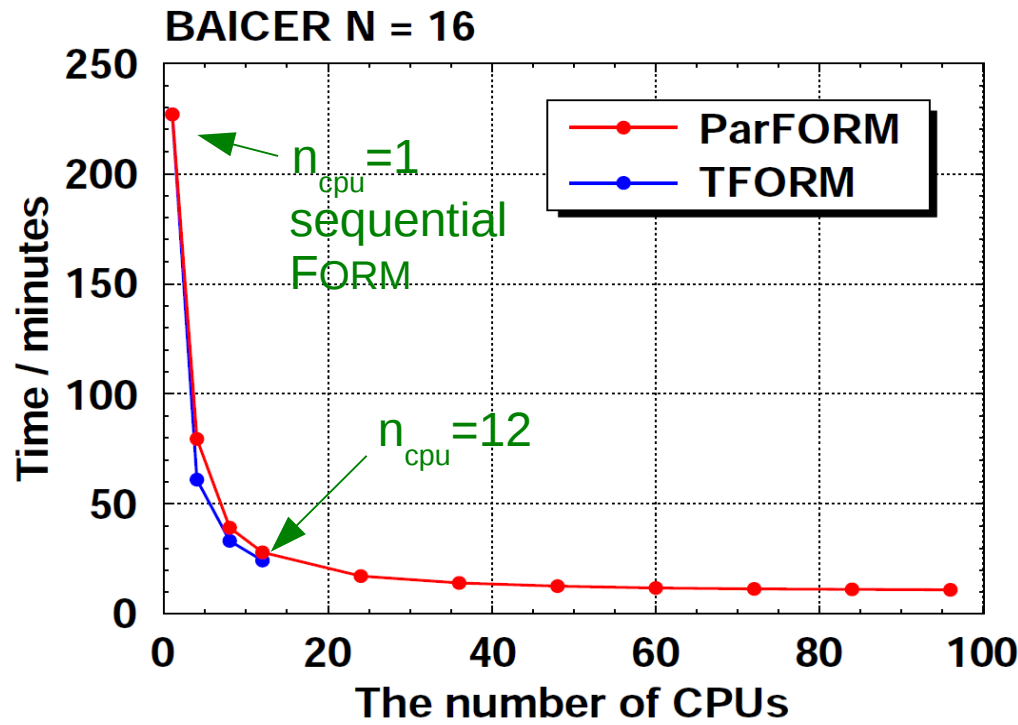
- Multithreaded version of FORM.
- Based on the POSIX threads (Pthreads).
- Communication via the shared memory space.
- Performance gain on multicore computers.

NIKHEF, 2005-

Tentyukov, Vermaseren '10

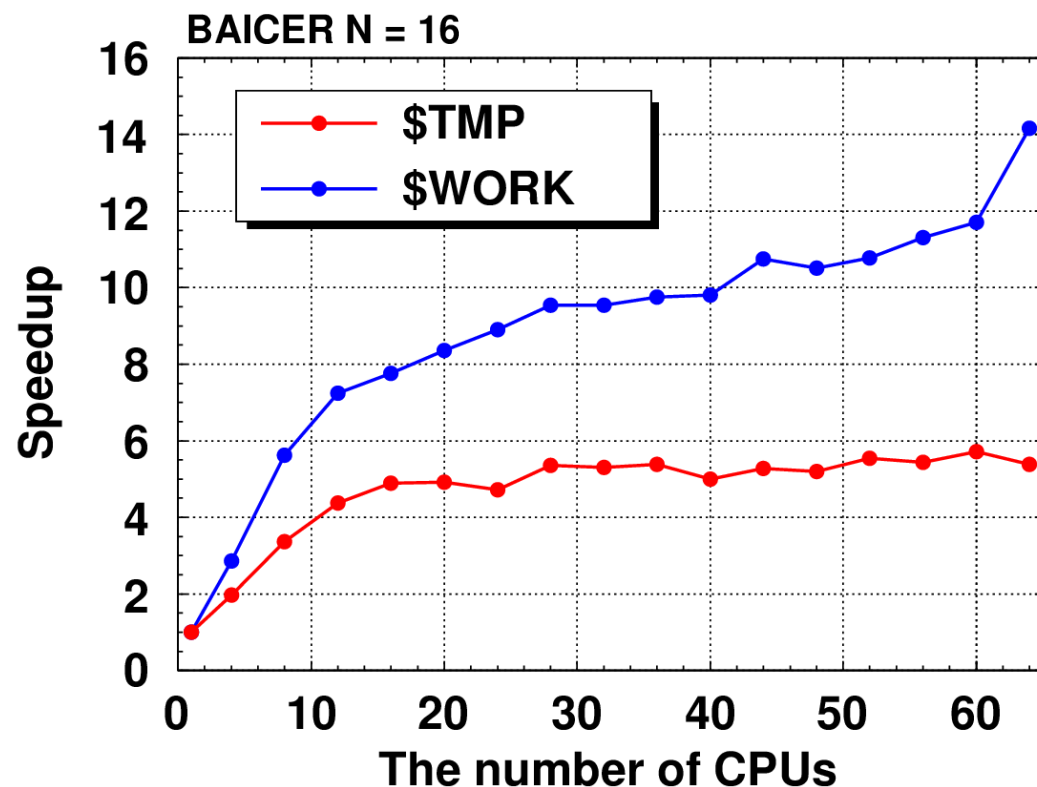
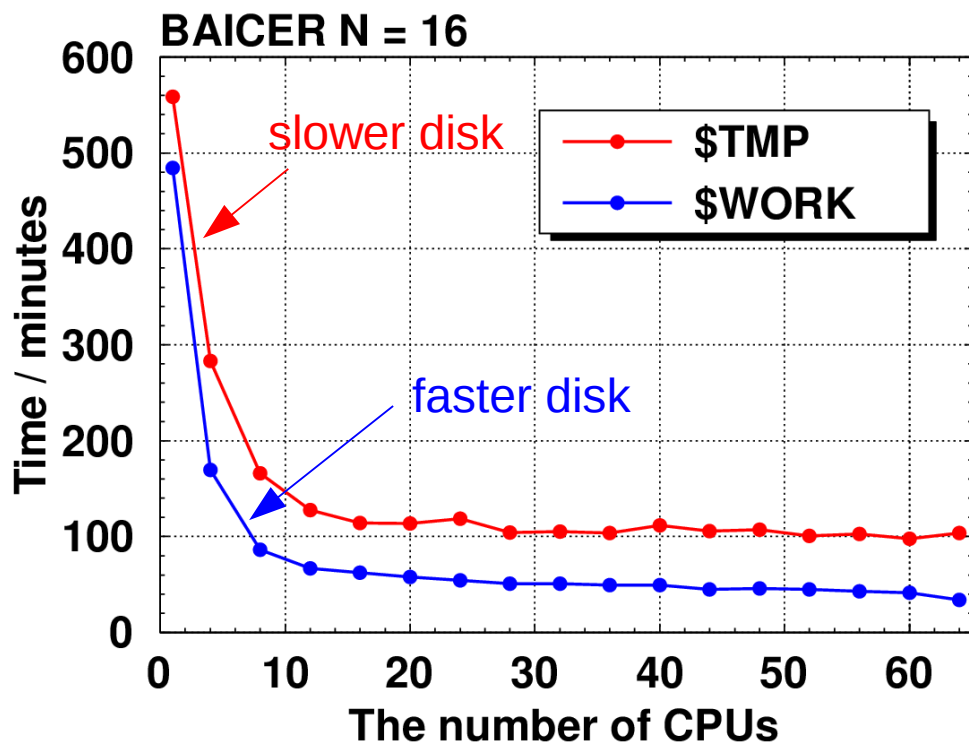


Benchmark I



- BAICER benchmark on ttpmoon cluster.
Each node has 12 cores (X5675 @ 3.07GHz),
96 GB RAM, 3.6TB local disk (Raid 0 with 6 stripes)
and connected by QDR Infiniband.

Benchmark II



- BAICER benchmark with ParFORM on HP XC4000 at KIT SCC. Each node has 4 cores (AMD Opteron @ 2.6GHz).

FORMTMP = \$TMP: local disk (R/W perf. / node: 60/60MB/s)

\$WORK: global disk (R/W perf. / node : 320/400MB/s)

Disk speed can considerably affect on the performance.

New Features in FORM 4

- Many features have been added since FORM 3.
 - Polynomial factorisation.
 - Rational functions as coefficients.
 - New statements, e.g., Transform statement.
 - Extra Symbols, ToPolynomial, FromPolynomial.
 - New functions (some of them are for polynomial algebra including `gcd_`, `div_`, `rem_`).
 - Checkpoints (recovery from a crash).
 - System independent save files.
 -

FORM 4: Factorisation

- The statement **FactArg** now factorise argument of functions.

```
Symbol x, a, b;  
CFunction f;  
Local E = f(a*b + x*b + x*a + x^2);  
FactArg f;  
Print;  
.end
```

E =
f(a + x, b + x);

- FORM 3 does not factorise this function argument.
(Only overall factors can be factorised.)
- Also factorisation of
 - Expressions : Factorize / Unfactorize
 - \$-variables : FactDollar / #FactDollar

FORM 4: Rational Functions as Coeffs.

- Normal sorting: $\frac{1}{2}x + \frac{1}{3}x \longrightarrow \frac{5}{6}x$
- PolyFun (already in Form 3): $\text{acc}(\frac{1}{2}+a)*x + \text{acc}(\frac{1}{2}+b)*x \longrightarrow \text{acc}(1+a+b)*x$
- Rational coefficients can be used with **PolyRatFun**.
- The first argument of the function serves as numerator and the second as denominator.

```
Symbol x,y,z;  
CFunction rat;  
PolyRatFun rat;  
Local E = x * rat(y,z) + x * rat(y,1-z)  
          + x^2 * rat(y^2-1,y-1);
```

```
Print;  
.end
```

E =

$x*\text{rat}(-y, z^2 - z) + x^2*\text{rat}(y + 1, 1);$

$$\frac{y}{z} + \frac{y}{1-z} = \frac{-y}{z^2 - z}$$

$$\frac{y^2 - 1}{y - 1} = y + 1$$

Application: MincerExact (1/3)

- Mincer (program for 3-loop massless propagator diagrams) works in expansions in $\epsilon = (4 - d)/2$, typically up to 6th power.
- Big tables for expansions of Pochhammer symbols etc. are needed.
- Using PolyRatFun, no need of expansions at all. Code is much cleaner/shorter and only slightly slower.
- Results for Mellin moments look like:

```
VALUE = GschemeConstants(0,0)^2*GschemeConstants(2,0)*  
cf^2*rat(-16+96*ep-48*ep^2-640*ep^3+1680*ep^4-  
1824*ep^5+944*ep^6-192*ep^7,9+33*ep+36*ep^2+12*ep^3);
```

Application: MincerExact (2/3)

- One can expand the rational function in the result around $\epsilon = 0$ up to any order you want.

```
Local E = rat(-16+96*ep-48*ep^2-640*ep^3+1680*ep^4  
              -1824*ep^5+944*ep^6-192*ep^7,  
              9+33*ep+36*ep^2+12*ep^3);
```

```
identify rat(x?,y?) = num(x)*den(y);
```

```
E = num(-16+96*ep-48*ep^2-640*ep^3+1680*ep^4  
        -1824*ep^5+944*ep^6-192*ep^7)  
    *den(9+33*ep+36*ep^2+12*ep^3);
```

```
factarg den;
```

```
E = num(-16+96*ep-48*ep^2-640*ep^3+1680*ep^4  
        -1824*ep^5+944*ep^6-192*ep^7)  
    *den(3, 1+ep, 1+2*ep, 3+2*ep);
```

Application: MincerExact (3/3)

```
E = num( -16+96*ep-48*ep^2-640*ep^3+1680*ep^4  
        -1824*ep^5+944*ep^6-192*ep^7 )  
      *den(3, 1+ep, 1+2*ep, 3+2*ep);
```

chainout den;

```
E = num( -16+96*ep-48*ep^2-640*ep^3+1680*ep^4  
        -1824*ep^5+944*ep^6-192*ep^7 )  
      *den(3)*den(1+ep)*den(1+2*ep)*den(3+2*ep);
```

- Expanding num and den up to $\mathcal{O}(\epsilon^6)$ gives

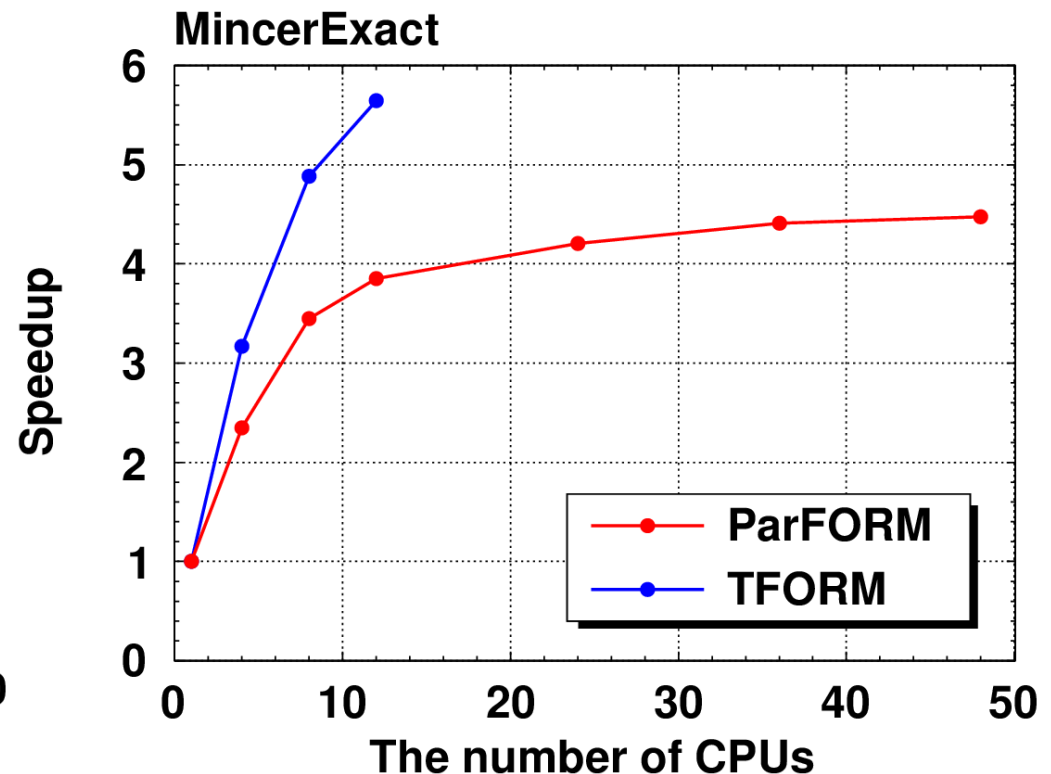
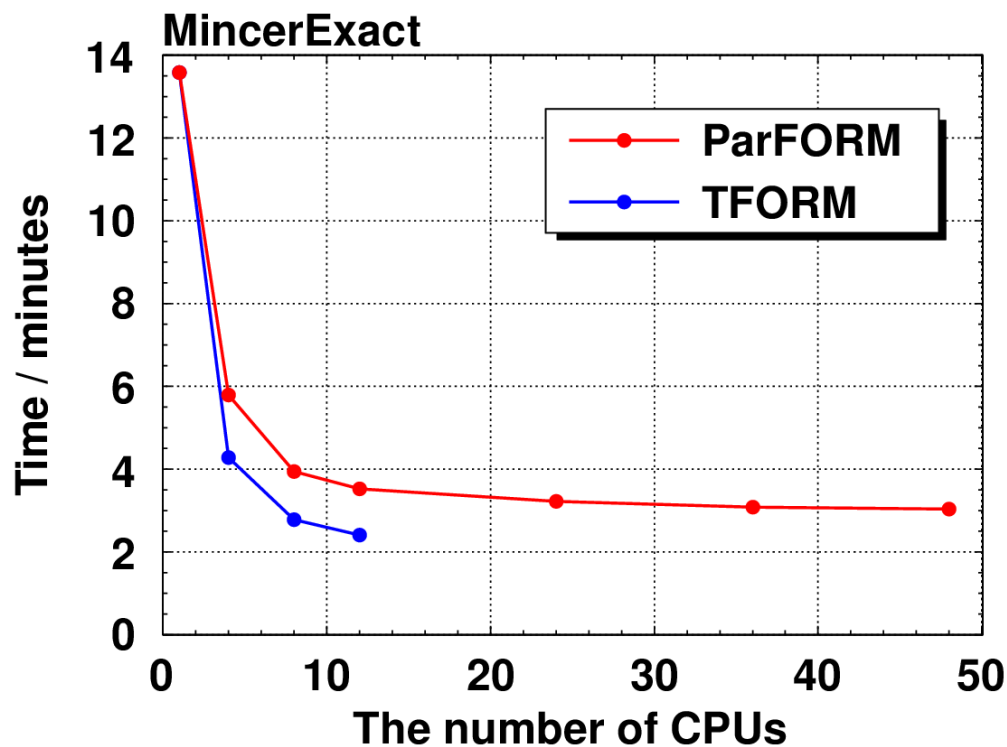
```
E = -16/9  
    +464/27*ep  
    -4960/81*ep^2  
    +21152/243*ep^3  
    +65264/729*ep^4  
    -1744048/2187*ep^5  
    +16761728/6561*ep^6;
```

ParFORM with New Features

- ParFORM is ready for the new features of the version 4 (and also TFORM).
- Some new features needed special code for MPI interactions between the master and the workers, for example:
 - Factorised \$-variable. (factorized subexpressions)
 - Factorised expressions. (sync. of some flags)
 - ...
- Users do not need to worry about the implementation. It is transparent for users.

MincerExact on Parallel FORM

- The benchmark result of MincerExact (calcdia.frm with keeping higher order gauge terms) on ttpmoon. Since the problem is not so big (14min by FORM), only small benefit. But ParFORM and TFORM work correctly.



\$-variables on Parallel Versions (1/2)

- The parallelisation is transparent to users: most of existing FORM programs can get a benefit without any modifications.
- \$-variables can obstruct parallel execution. In the previous example, FORM does not know how to combine the results of \$n on the workers. Enter the sequential mode.

```
Symbol x,n;  
Local expr = (1+x)^5;  
  
#$n = 0;  
if ( count(x,1) > $n ) $n = count_(x,1);  
.sort  
  
#message the maximum power of x is ` $n '  
.end
```


\$-variables on Parallel Versions (2/2)

- Give a hint:

```
ModuleOption maximum $n;
```

Then FORM can run in the parallel mode and combine \$-variables correctly after the execution of the module.

```
Symbol x,n;  
Local expr = (1+x)^5;  
  
#$n = 0;  
if ( count(x,1) > $n ) $n = count_(x,1);  
ModuleOption maximum $n;  
.sort  
  
#message the maximum power of x is `'$n'  
.end
```

- Other hints:

- local
- maximum
- minimum
- sum

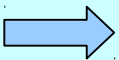
Parallel Versions of FORM in CVS (1/2)

- FORM is now **open source**. One can download the source code from the FORM CVS repository:
<http://www.nikhef.nl/~form/formcvsv.php>
(alternatively, a Git Mirror: <http://github.com/tueda/form>)
- ParFORM/TFORM sources are also in the repository.
- One can find pre-compiled FORM/TFORM executables on the web site. But one may want to build them because
 - Bug fixes in the latest CVS version, (sometimes more bugs...)
 - Compiler optimisations for your machine.
- For ParFORM, one has to build it on his/her environment because MPI implementations are not binary-compatible.

Parallel Versions of FORM in CVS (2/2)

- To build executable files, automatic configuration is available for UNIX-like environments, e.g., Linux.

```
$ wget http://www.nikhef.nl/~form/formcvsv.php?dl=
formcvsv.tar.gz -O formcvsv.tar.gz
$ tar xzf formcvsv.tar.gz
$ cd formcvsv
$ autoreconf -i
$ ./configure --enable-parform
$ make
```

 form, tform, parform in sources subdirectory
(or one can “make install”)

- Run as

```
$ form myprogram.frm
$ tform -w 4 myprogram.frm
$ mpirun -np 16 parform myprogram.frm
```

Summary

- FORM 4.0 is now available.

FORM version 4.0, J. Kuipers, TU, J.A.M. Vermaseren,
and J. Vollinga, arXiv:1203.6543 [cs.SC].

- Many new features, e.g., polynomial algebra.
 - Future calculations may use significantly different algorithms from those used in the past.
-
- ParFORM and TFORM are ready for the new features.
 - No modification in one's program is needed.
 - One can try the parallel versions for one's big problems.

<http://www.nikhef.nl/~form/>

Backup Slides

Preprocessor

- Powerful preprocessor to prepares/filters the input to be passed to the compiler, e.g., flow control.

- Preprocessor instructions.

- Preprocessor variables.

```
#define flag "1"  
#define N "3"
```

```
#ifdef `flag'  
    #write "`N'"  
#endif  
.end
```

3

```
#procedure proc(a,b)  
    #write "`a'+`b'={`a'+`b'}"  
#endprocedure
```

```
#do i=1,3  
    #call proc(`i',2)  
#enddo  
.end
```

1+2=3

2+2=4

3+2=5

\$-variables

- A (small) expression stored in the memory.
- Get/set in both preprocessor and processor phases.

```
Symbol x, a, b;  
Cfunction f;  
Local E = f(a+b) + f(a+2*b);  
.sort  
id f(x?$x) = f(x);  
multiply $x;  
Print;  
.end
```

```
#$a = 1;  
$a = 1;  
.sort  
#write "`$a'"  
multiply $a;  
.end
```

```
E =  
    f(b + a)*b + f(b + a)*a + 2*f(2*b + a)*b  
+ f(2*b + a)*a;
```

Polynomial Algebra in FORM 4

- Programmed by Jan Kuipers.
- Polynomial algebra in FORM 4 basically consists of:
 - Greatest common divisor
 - Factorization
 - PolyRatFun

Polynomial Manipulation

- Distributed degree sparse and variable dense representation of polynomials is used:

$$p = \sum_{\text{terms}} ax^i y^j z^k$$

- Stored as an array of pairs of coefficient and exponents $(a; i, j, k)$, $a \neq 0$, $i, j, k = 0, 1, 2, \dots$
- Needs conversions from FORM expressions, but is faster.
- Fast algorithms for multiplying and division using heaps are implemented.

Greatest Common Divisor

- The function **gcd_** gives the greatest common divisor of its arguments. The arguments can be multivariate polynomials in FORM 4.

```
Symbol x,y;  
Local E = gcd_(x^2+x*y, y^2+x*y);  
Print;  
.end
```

```
E =  
y + x;
```

- FORM 3 would gives $E = 1$.

Greatest Common Divisor Algorithm

- For small polynomials, a heuristic that substitutes integers and performs integer gcd calculations is used.
- For large polynomials, Zippel's modular algorithm is used.
- Speed is comparable to Mathematica.

Factorization of Dollar Variables

- The preprocessor instruction **#FactDollar** factorizes a dollar variable.

```
Symbol x,y;  
#$a = x^2-y^2;  
#FactDollar $a;
```

```
#do i=1,`$a[0]`  
    #write "%$", $a[`i'];  
#enddo  
.end
```

```
-y+x  
y+x
```

- $a[0]$ stores the number of factors.
- $a[1], \dots, a[a[0]]$ store the factorized parts.

Factorization of Dollar Variables (cont'd)

- Analogous statement **FactDollar** for runtime factorization.

```
Symbol x,y;  
CFunction f;  
Local E = 1;  
$a = x^2-y^2;  
FactDollar $a;  
  
do $i=1, $a[0];  
    multiply f($a[$i]);  
enddo;  
Print;  
ModuleOption local $i, $a;  
.end
```

```
E =  
    f( - y + x)*f(y + x);
```

Factorization Algorithm

- For univariate polynomials Berlekamp's algorithm is used.
- Multivariate polynomials are reduced to univariate and afterwards Hensel lifting is used to reconstruct multivariate factors.
- To factorize this polynomial

**-6272714818668017*a^35*b^22*c^20*d^9*e^21
-6867348605700329*a^34*b^33*c^19*d^11*e^36
+323798222821062*a^34*b^20*c^29*d^8*e^18
+ (... 10 more terms ...)
+2081169781417560*a^28*b^10*c^13*d^27*e^12
-285878431480222*a^28*b^4*c^25*d^13*e^13
-520827763173144*a^27*b^4*c^19*d^24*e^11**

FORM takes 9 sec and Mathematica takes 900 sec.

Application: MincerExact (1/5)

- Mincer (program for 3-loop massless propagator diagrams) works in expansions in $\epsilon = (4 - d)/2$, typically up to 6th power.
- Big tables for expansions of Pochhammer symbols and alike are needed.
- Using PolyRatFun, no need of expansions at all. Code is much cleaner/shorter and only slightly slower.
- Results for Mellin moments look like:

```
VALUE=GschemeConstants(0,0)^2*GschemeConstants(2,0)*  
cf^2*rat(-192*ep^7+944*ep^6-1824*ep^5+1680*ep^4-640*  
ep^3-48*ep^2+96*ep-16,12*ep^3+36*ep^2+33*ep+9)
```

Application: MincerExact (2/5)

- Let's have a closer look at an answer of MincerExact.

First, factorize the denominator:

```
Symbol ep, a, b, c, d;  
CFunction rat, num, den;  
Local E = rat(-192*ep^7+944*ep^6-1824*ep^5+1680*ep^4  
              -640*ep^3-48*ep^2+96*ep-16,  
              12*ep^3+36*ep^2+33*ep+9);  
id rat(a?, b?) = num(a)*den(b);  
FactArg den;  
ChainOut den;  
id den(a?number_) = 1/a;  
Print +s;  
.sort
```

E=

```
+1/3*num(-16+96*ep-48*ep^2-640*ep^3+1680*ep^4  
          -1824*ep^5+944*ep^6-192*ep^7)  
*den(1+ep)*den(1+2*ep)*den(3+2*ep);
```


Application: MincerExact (3/5)

- Make a partial fraction expansion:

```
SplitArg den;  
FactArg den;  
id den(a?, ep, b?) = 1/b*den(a/b, ep);  
repeat id den(a?, ep)*den(b?, ep) =  
    (den(a, ep)-den(b, ep)) / (b-a);  
Print +s;  
Bracket num;  
.sort
```

$$\frac{1}{a + b\epsilon} = \frac{1}{b} \frac{1}{a/b + \epsilon}$$

$$\frac{1}{a + \epsilon} \frac{1}{b + \epsilon} = \frac{1}{b - a} \left(\frac{1}{a + \epsilon} - \frac{1}{b + \epsilon} \right)$$

E=

```
+num( -16+96*ep-48*ep^2-640*ep^3+1680*ep^4  
      -1824*ep^5+944*ep^6-192*ep^7)*(  
      +1/6*den(1/2, ep)  
      +1/6*den(3/2, ep)  
      -1/3*den(1, ep)  
      );
```

Application: MincerExact (4/5)

- Rewrite it once more:

```
id num(a?) = a;  
repeat id ep*den(a?, ep) = 1 - a*den(a, ep);  
Print +s;  
.sort
```

$$\frac{\epsilon}{a + \epsilon} = 1 - \frac{a}{a + \epsilon}$$

E=

```
-7828/3  
+3803/3*ep  
-488*ep^2  
+380/3*ep^3  
-16*ep^4  
+243/8*den(1/2, ep)  
+153125/24*den(3/2, ep)  
-5120/3*den(1, ep)  
;
```

Application: MincerExact (5/5)

- Finally, expand around $\epsilon = 0$ up to 6th order

```
Symbol ep(:6);  
repeat id den(a?,ep) = 1/a - ep/a * den(a,ep);  
Print +s;  
.end
```

$$\frac{1}{a + \epsilon} = \frac{1}{a} - \frac{\epsilon}{a} \frac{1}{a + \epsilon}$$

E=

```
-16/9  
+464/27*ep  
-4960/81*ep^2  
+21152/243*ep^3  
+65264/729*ep^4  
-1744048/2187*ep^5  
+16761728/6561*ep^6  
;
```

- Same result as Mincer.

ParFORM with New Features

- Some new features needed special code for MPI interactions between the master and the workers, for example:
 - Factorized $\$$ -variable. (factorized stuff)
 - Factorized expressions. (some flags)
 - FromPolynomial in parallel module. (conversion table made by ToPolynomial)
 - `#inside / #endinside` construction containing RHS expressions. ($\$$ -variables, redefined preprocessor variables)
- Users do not need to worry about the implementation.

An Example: RHS expressions at Compile-Time

- Even at compile-time, one can manipulate \$-variables by using `#Inside` preprocessor instruction. Since the master and all workers compile the user program and \$-variables can affect the compilation, all processes must know the result.

`#Inside` can contain any regular statements.

If expressions appear on the right hand side of substitutions at compile-time, the master broadcasts the result to all the workers.

```
#inside $a
  id x^2 = F;
  id f(x?$b) = 1;
  $c = 1;
  redefine var "1";
#endinside
```

An expression defined previously
(stored on the master; the workers can't access it)

} Change on \$b and \$c.

← Change on preprocessor variable var.

- In this case, \$a, \$b, \$c and var will be broadcast.

Comparison with Other CAS

Mathematica, Maple, etc.



Swiss Army knife

FORM



Chef's knife

- Much built-in mathematical knowledge (integration, solving equations, special functions etc.)
- Very general, versatile (sometimes overkill)
- Big and slow (especially on large problems)
- (Many of them are) proprietary
- Limited built-in knowledge (calculus with tensors and gamma matrices, etc.)
- Optimized for efficiency
- Small and fast (also on large problems)
- Open source

Modulus Calculus in FORM 4

- Various options in **Modulus** statement has been added and the syntax has been changed slightly from the previous one.
 - PlusMin, Positive
 - AlsoFunctions, NoFunctions
 - CoefficientsOnly
 - AlsoDollars, NoDollars
 - InverseTable, NoInverseTable
 - AlsoPowers, NoPowers
 - PrintPowersOf

Recovery mechanism

- Programmed by Jens Volinga.
- A mechanism allowing the user to make snapshots of runtime information, and recovery from them when unforeseen machine failures occurs.

On Checkpoint;

Snapshot FORMrecv.tmp created at the each end of module

form -R myprogram.frm

Recovery from the snapshot

- More options are in the manual.

Extra Symbols

- A mechanism to replace non-symbol objects by internally generated symbols.

```
Symbol a,b;  
CFunction log;  
Local E = (log(a)+log(b))^2;  
Print;  
.sort
```

$$E = \log(a)^2 + 2*\log(a)*\log(b) + \log(b)^2;$$

```
ExtraSymbols array, Y;  
ToPolynomial;  
Print;  
.sort
```

$$E = Y(1)^2 + 2*Y(2)*Y(1) + Y(2)^2;$$

- Can be used for code generations (C or Fortran, etc.)

Transform Statement

- Manipulation of function arguments. allows speedy transformations without the need of multiple statements or repeat loops (e.g., ArgExplode ArgImplode).

```
Symbol x,x1,x2;
Cfunction H,H1;
Local E = H(3,4,2,6,1,1,1,2);
repeat id H(?a,x?!{0,1},?b)
    = H(?a,0,x-1,?b);
Print;
.sort
multiply H1;
repeat id H(x?,?a)*H1(?b)
    = H(?a)*H1(?b,1-x);
id H1(?a)*H = H(?a);
Print;
.sort
repeat id H(x1?,x2?,?a)
    = H(2*x1+x2,?a);
Print;
.end

Cfunction H;
Local E = H(3,4,2,6,1,1,1,2);
transform,H,explode(1,last),
    replace(1,last)=(0,1,1,0),
    encode(1,last):base=2;
Print;
.end

E =
    H(907202);
```

Short List of Bug Fixes on ParFORM

- Fixed bugs about RHS expressions:
 - The output routine can override the buffers storing RHS expressions.
 - Memory bugs when RHS expressions are broadcast.
 - Didn't work with “Keep Brackets” statement.
- InParallel + Keep Brackets didn't work.
- The handling of “dummy indices” was not sufficient.
- Creation of “bracket index” in parallel module was broken.
- ...

Support for New Features

- The following features needed some modifications of ParFORM source.
 - IntoHide statement.
 - Initialization of the random number generator.

Support for New Features (cont'd)

- The following features needed special code for MPI interactions between the master and the workers.
 - Factorized \$-variable. (factorized stuff)
 - Factorized expressions. (some flags)
 - FromPolynomial in parallel module. (conversion table made by ToPolynomial)
 - #inside / #endinside construction containing RHS expressions. (\$-variables, redefined preprocessor variables)

RHS expressions (Run-time)

- Expression names appearing in the right-hand side.

Symbol a, b, x;

Local F = a + b;
Local G = x + F;

Symbol a, b, c, d;

Local F = a + b;
Local G = c + d;
id a = G;

- Substitutions to **F** / **G** are performed in workers, but only the master knows whole expressions of **F** / **G**.
- The whole expressions of **F** / **G** must be broadcast from the master to the (all) slaves before executing the module.

\$-variable

- Variables which store small expressions and can be accessed from both the preprocessor (compile-time) and the virtual machine (run-time).

compile-time

```
#$a = 1+x;  
id a = ` $a ';
```

Expanded as a preprocessor variable at the compilation.

run-time

```
$a = 1+x;  
id a = $a;
```

Repeated for each term in active expressions.

- Substitution by pattern matching

```
id f(x?$a) = whatever;
```

- Hint for parallelization:

```
ModuleOption local $a;
```

Inside Statement (Run-time)

- Statements in **Inside ... EndInside** are executed for the given $\$$ -variable(s), not for active expressions.

```
$a = (1+x)^5;
```

```
inside $a;
```

```
    id x^2 = 1;
```

```
    if (count(x,1)) discard;
```

```
    multiply 2;
```

```
endinside;
```

} Operated on terms in \$a.

- If RHS expressions are in **Inside ... EndInside**, they have to broadcast before executing the module, as usual.

#Inside Instruction (Compile-time)

- Statements in **#Inside ... #EndInside** are executed for the given $\$$ -variable(s) **in compile-time**.

```
#$a = (1+x)^5;
```

```
#inside $a
```

```
  id x^2 = 1;
```

```
  if (count(x,1)) discard;
```

```
  multiply 2;
```

```
#endinside
```

} Operated on terms in \$a.

RHS expressions (Compile-time)

- At compile-time, the master and all workers executes manipulations on $\$$ -variables, because they are needed on all processes for the compilation.

```
 $\$a$  = 1;                                #inside $a  
                                         id x = 1;  
                                         #endinside
```

- RHS expressions can appear in substitution at compile-time. Only the master knows the whole expression.

```
 $\$a$  = F;                                #inside $a  
                                         id x = F;  
                                         #endinside
```

- Solution: In such cases, only the master executes the manipulations, and then the master broadcasts the result to the all workers.

RHS expressions (Compile-time)

- A complication: `#Inside ... #EndInside` can contain *all* executable statements.

```
#inside $a
  id x^2 = F;
  id f(x?$b) = 1;
  $c = 1;
  redefine var "1";
#endinside
```

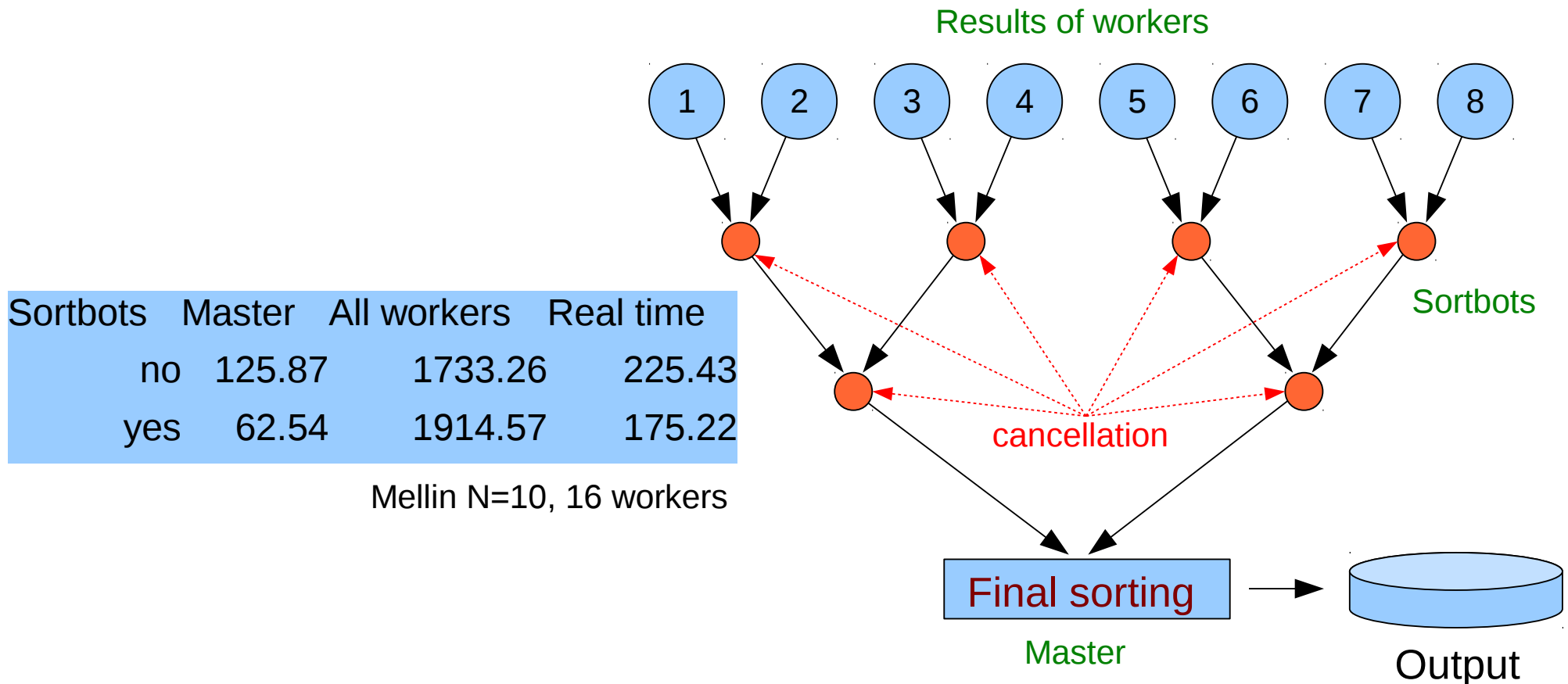
} Change on \$b and \$c.

← Change on preprocessor variable var.

- In this case, \$a, \$b, \$c and var must be broadcast.

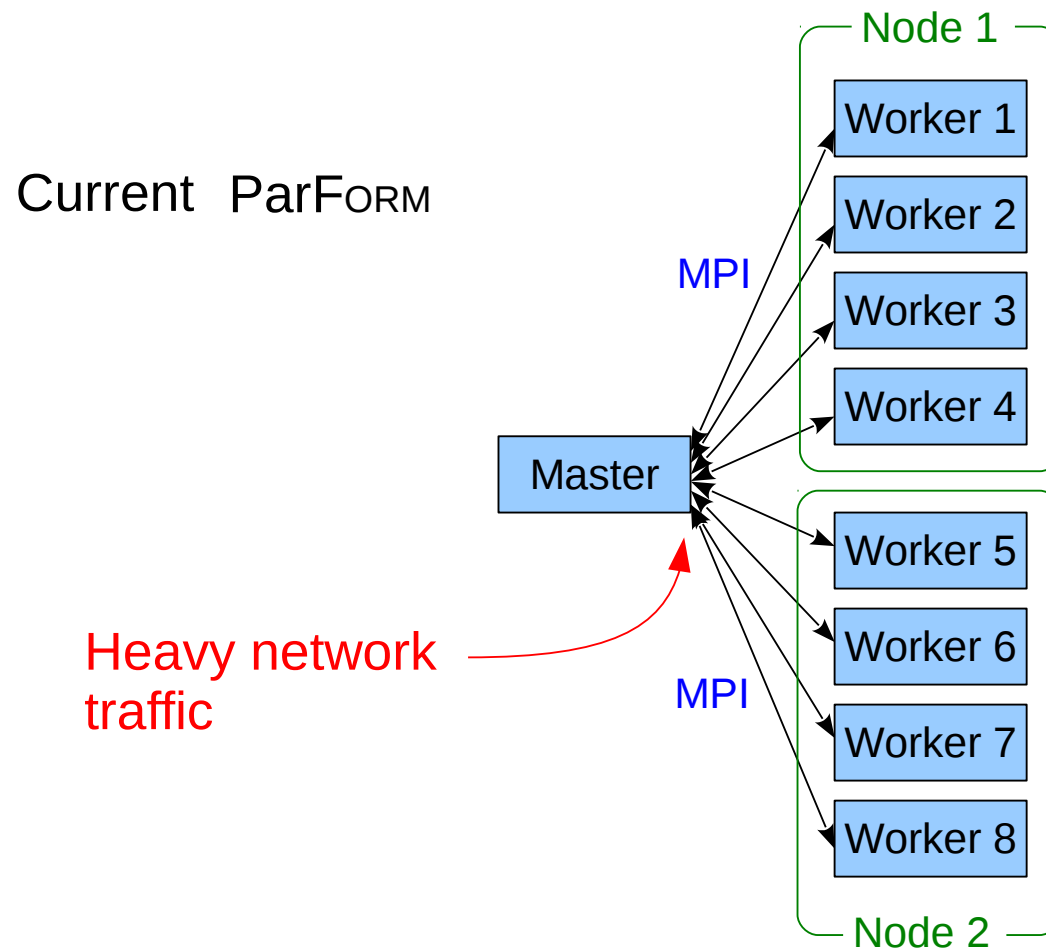
Sortbots (TFORM)

- The final sorting is a bottleneck.
- Special threads (sortbots) merge each two results.



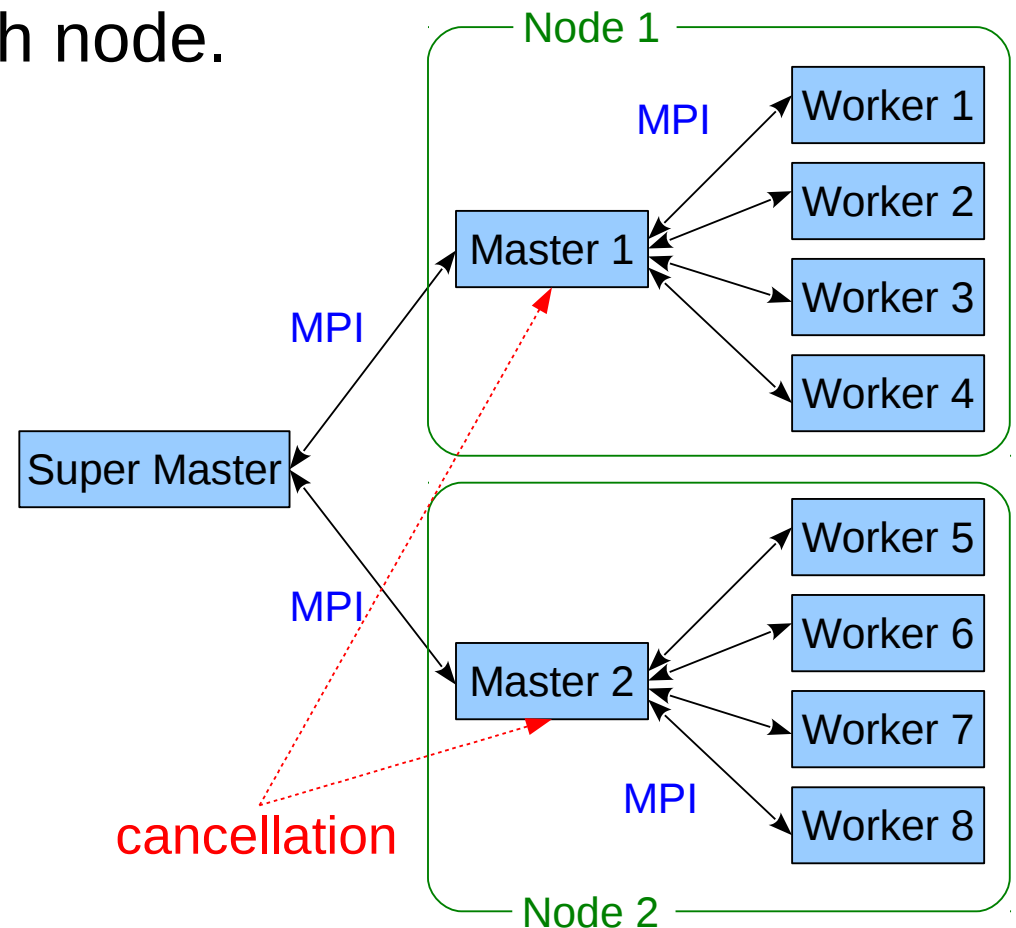
Future Plans of FORM Parallelisation

- On computer clusters built from multicore processors:
 - Heavy network traffic to the master.



Future Plans of FORM Parallelisation

- On computer clusters built from multicore processors:
 - Each node has its own master.
 - Still MPI overheads in each node.



Future Plans of FORM Parallelisation

- On computer clusters built from multicore processors:
 - Hybrid MPI/Pthreads parallelisation.
 - Avoid heavy network traffic to the master.
 - No MPI overheads in each node.

